Program Testing and Analysis: Specification Mining

Dr. Michael Pradel
Software Lab, TU Darmstadt

Partly based on slides from Michael Ernst, Andreas Zeller, and Andrzej Wasylkowski
Outline

1. Introduction
2. Program Invariants
3. Finite State Machines
4. Programming Rules

Mostly based on these papers:

- *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, Ernst et al., IEEE TSE, 2001
- *Mining Specifications*, Ammons et al., POPL, 2002
Formal Specifications

- Formal, mathematical description of the intended behavior of a program

Examples:

- Pre- and post-conditions:
  ```javascript
  // pre: typeof(x) === "number"
  function abs(x) { ... }
  // post: typeof(ret) === "number" &&
  // ret >= 0
  ``

- Finite-state machines:
Uses of Specifications

Traditionally, mainly used for formal verification

- Demonstrate that program is correct w.r.t. its specification
- Mathematical proof
- Ideally, static verification
  - Avoid running an incorrect program
- Also: Runtime verification
  - Detect and potentially prevent problems when they happen
The Problem

So why not formally specify and verify all software?

■ Huge effort
■ Completely specifying a large system is practically impossible
■ Complex specification is likely to have mistakes
■ In practice:
  □ Used mostly for safety critical systems
  □ Used only to specify important properties (e.g., no crash)
Specification Mining

■ Idea: Infer specifications from existing software
  □ No human effort
  □ Get specifications "for free"

■ Examples:
  □ Pre- and post-conditions:
    Analyze function and check which properties the inputs and outputs fulfill
  □ Finite state machines:
    Analyze code and identify its states and transitions between them
Wait a Minute ...

- How to validate that a program is correct by inferring the specification from the program itself?
- Sounds contradictory, but it works
  - Infer common behavior, report anomalies as potential bugs
  - Infer specifications from one code base, use them to check another
    * Different programs that use the same API
    * Different versions of the same program
  - Detect inconsistencies in the code itself (non-null assumption vs. null check)
Uses of Mined Specifications

- **Software evolution**
  - Understand behavior of program
  - Generate documentation
  - Use as oracle for regression testing

- **Anomaly detection**
  - Outliers are potential bugs

- **Support formal specification of an existing system**
  - Starting point for full specification
Outline

1. Introduction
2. Program Invariants
3. Finite State Machines
4. Programming Rules

Mostly based on these papers:

- *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, Ernst et al., IEEE TSE, 2001
- *Mining Specifications*, Ammons et al., POPL, 2002
Program Invariants

- Invariant = Data property that holds in all runs
  - At entry of f(), x is an odd number
  - \(0 \leq y \leq 10\)

- Useful in software development
  - Protect programmers from making erroneous changes
  - Verify properties of a program

- Can be explicitly stated in programs
  - Programmers can annotate code with invariants
  - Huge effort
  - Important invariants may be missed
Example

```javascript
function sumArray(b, n) {
    var i = 0, s = 0;
    while (i !== n) {
        s += b[i];
        i++;
    }
    return s;
}
```
Example

Some invariants from running with 100 randomly generated inputs of length 7-13:

- **Pre-conditions:**
  - $n = b.length$
  - $7 \leq n \leq 13$

- **Post-conditions:**
  - $n = i = b.length$
  - $b = orig(b)$
  - $s = sum(b)$

- **Loop invariants**
  - $n = b.length$
  - $0 \leq i \leq 13$
  - $s = sum(b[0..i - 1])$
Daikon Invariant Detector

- **Dynamic analysis**: Infers invariants from particular execution
- **Step 1**: **Instrument** source code
  - Trace variables of interest
- **Step 2**: **Run** instrumented program using test suite
- **Step 3**: **Infer invariants** from instrumented and derived variables
Step 1: Instrumentation

Insert instrumentation points
- Function entry
- Function exit
- Loop heads

Write to a file values for
- all variables in scope
- global variables
- function arguments
- local variables
- function’s return value
Step 2: Execution

- Instrumented program writes file with runtime values
- Result: Trace of execution
Step 3: Inference

Daikon has library of invariant patterns over variables (e.g., $x, y, z$) and constants (e.g., $a, b, c$), e.g.:

- Check for each variable:
  - Constant or small number of values
- Check for numeric variables:
  - Range: $a \leq x \leq b$
- Check for multiple numbers:
  - Set of functions, e.g., $x = \text{abs}(y)$
  - Comparisons, e.g., $x < y$
- Check for sequences:
  - Sortedness
Step 3: Inference

Daikon has library of invariant patterns over variables (e.g., $x, y, z$) and constants (e.g., $a, b, c$), e.g.:

- Check for each variable:
  - Constant or small number of values

- Check for numeric variables:
  - Range: $a \leq x \leq b$

- Check for multiple numbers:
  - Set of functions, e.g., $x = \text{abs}(y)$
  - Comparisons, e.g., $x < y$

- Check for sequences:
  - Sortedness

Only matching patterns are preserved
What post-conditions could Daikon infer from tests $g(1)$ and $g(3)$?

```javascript
function g(n) {
    var x = n * 2;
    var y = 0;
    for (var i = 0; i < x; i++) {
        y += i;
    }
    return y;
}
```

$x < y \quad i \geq 2 \quad 1 \leq y \leq 15 \quad n = 1$
What post-conditions could Daikon infer from tests $g(1)$ and $g(3)$?

```javascript
function g(n) {
    var x = n * 2;
    var y = 0;
    for (var i = 0; i < x; i++) {
        y += i;
    }
    return y;
}
```

$x < y$  

\[
\begin{align*}
    i & \geq 2 \\
    1 & \leq y \leq 15 \\
    n & = 1
\end{align*}
\]